Design By Contract
Deontic Design Language for Multiagent Systems

Christophe Garion[1] and Leendert van der Torre[2]

[1] SUPAERO
10 avenue Édouard Belin
31055 Toulouse
France
garion@supaero.fr
[2] University of Luxembourg
Luxembourg
leendert@vandertorre.com

Abstract. Design by contract is a well known theory that views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by assertions. However, there is a gap between this theory and software engineering concepts and tools. For example, dealing with contract violations is realized by exception handlers, whereas it has been observed in the area of deontic logic in computer science that violations and exceptions are distinct concepts that should not be confused. To bridge this gap, we propose a software design language based on temporal deontic logic. Moreover, we show how preferences over the possible outcomes of a supplier can be added. We also discuss the relation between the normative stance toward systems implicit in the design by contract approach and the intentional or BDI stance popular in agent theory.

1 Introduction

Design by contract [1–3] is a well known software design methodology that views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by assertions. It has been developed in the context of object oriented programming, it is the basis of the programming language Eiffel, and it is well suited to design component-based and agent systems. However, there is still a gap between this methodology and formal tools supporting it. For example, dealing with contract violations is realized by exception handlers, whereas it is well known in the area of deontic logic in computer science [4, 5] that violations and exceptions are distinct concepts that should not be confused. Formal tool support for design by contract is therefore a promising new application of deontic logic in computer science [6]. In this paper we study how extensions of deontic logic can be used as a design language to support design by contract. We address the following four research questions.

1. Which kind of deontic logic can be used as a design language to support design by contract?

2. How can we add preferences over possible outcomes of a routine?
3. What kind of properties can be formalized by such a design logic?
4. How does this approach based on deontic logic compare to the BDI approach, dominant in agent oriented software engineering?

The motivation of our work is the formal support for agent based systems. Recently several agent languages and architectures have been proposed which are based on obligations and other normative concepts instead (or in addition to) knowledge and goals (KBS), or beliefs, desires and intentions (BDI). In artificial intelligence the best known of these normative approaches is probably the IMPACT system developed by Subrahmanian and colleagues [7]. In this approach, wrappers built around legacy systems are based on obligations. We are interested in particular in designing component based agent systems such as agents based on the BOID architecture [8]. Notice that this paper does not address pure logical aspect. We do not define a specific logic for reasoning about such notions, but we use existing formalisms to model design by contract and preferences about possible outcomes of a routine.

The layout of this paper is as follows. In Section 2 we discuss design by contract, the deontic design language and contract violations. In Section 3 we introduce preferences over outcomes. In section 4 we compare this approach based on deontic logic to the KBS/BDI approach.

2 Design by contract

We explain design by contract by an example program in the Eiffel programming language. The explanation of design by contract as well as the example have been taken from [9]. For further details on design by contract, see [1–3].

2.1 Conditional obligations

Design By Contract views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by *assertions*. These assertions play a central part in the Eiffel method for building reliable object-oriented software. They serve to make explicit the assumptions on which programmers rely when they write software elements that they believe are correct. In particular, writing assertions amounts to spelling out the terms of the *contract* which governs the relationship between a routine and its callers. The precondition binds the callers; the postcondition binds the routine.

The Eiffel class in the left column of Figure 1 illustrates assertions (ignore for now the right column). An account has a balance (an integer) and an owner (a person). The only routines – **is** ... **do** ... **end** sequences – accessible from the outside are increasing the balance (deposit) and decreasing the balance (withdraw). Assertions play the following roles in this example.

Routine preconditions express the requirements that clients must satisfy when they call a routine. For example the designer of *ACCOUNT* may wish to permit a withdrawal operation only if it keeps the account's balance at or above the minimum. Preconditions are introduced by the keyword **require**.

```
class ACCOUNT
feature
 balance: INTEGER
 owner: PERSON
 min_balance: INTEGER is 1000
 deposit(sum:INTEGER) is
  require
   sum >= 0                              -- Ocr(sum >= 0)
  do
   add(sum)
  ensure balance = old balance + sum     -- Orc(balance = old balance + sum)
  end
 withdraw(sum:integer) is
  require
   sum >= 0                              -- Ocr(sum >= 0)
   sum <= balance - min_balance          -- Ocr(sum <= balance - min_balance)
  do
   add(-sum)
  ensure
   balance = old balance - sum           -- Orc(balance = old balance - sum)
  end
feature [NONE]
 add(sum:INTEGER) is
  do
   balance:=balance+sum
  end
invariant
 balance >= min_balance                 -- Or(balance >= min_balance)
end -- class ACCOUNT
```

Fig. 1. Class $ACCOUNT$

Routine postconditions, introduced by the keyword **ensure**, express conditions that the routine (the supplier) guarantees on return, if the precondition was satisfied on entry.

A class invariant must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class. The invariant appears in a clause introduce by the keyword **invariant**, and represents a general consistency constraint imposed on all routines of the class.

2.2 Deontic design language

We are interested in a deontic design language to support specification and verification based on design by contract. The deontic design language is therefore a kind of specification and verification language.

Syntactically, assertions are boolean expressions. To formalize the assertions in our design language, we use a deontic logic based on directed obligations, as used in electronic commerce and in artificial intelligence and law [10–13]. A modal formula $O_{a,b}(\phi)$ for a, b in the set of objects (or components, or agents) is read as "object a is obliged toward object b to see to it that ϕ holds". We write c and r for the caller and for the routine, such that the assertions in the program can be expressed as the logical formulae given in the right column in Figure 1. Summarizing:

Require ϕ = $O_{c,r}(\phi)$: caller c is obliged toward routine r to see to ϕ.
Ensure ϕ = $O_{r,c}(\phi)$: routine r is obliged toward caller c to see to ϕ.
Invariant ϕ = $O_r(\phi)$: routine r is obliged to see to ϕ.

To use the obligations above in a deontic design language, we have to add temporal information. First, we have to formalize "**old** *expression*" as it occurs for example in line 12 of class $ACCOUNT$. This expression is only valid in a routine postcondition, and denotes the value the expression has on routine entry. Consequently, we have to distinguish between expressions true at entry of the routine and at exit of it. More generally, we have to reason how the assertions change over time. For example, the require obligation only holds on entrance, the **ensure** obligation holds on exit, and the invariant obligation holds as long as the object exists. The obligations only hold conditionally. For example, if the preconditions do not hold, than the routine is not obliged to see to it that the ensure expression holds. Finally, the conditional obligations come into force once the object is created, and cease to exist when the object is destructed.

We therefore combine the logic of directed obligations with linear time logic (LTL), well known in specification and verification [14]. There are many alternative temporal logics which we could use as well. For example, in [15] deontic logic is extended with computational tree logic in BDIO$_{\mathrm{CTL}}$, and in [16] it is extended with alternating time logic (ATL). Semantics and proof theory are straightforward, see for example [15].

Definition 1 (Syntax O$_{\mathrm{LTL}}$). *Given a finite set A of objects (or components, or agents) and a countable set P of primitive proposition names. The admissible formulae of O_{LTL} are recursively defined by:*

1 *Each primitive proposition in P is a formula.*
2 *If α and β are formulae, then so are $\alpha \wedge \beta$ and $\neg\alpha$.*
3 *If α is a formula and $a, b \in A$, then $O_{a,b}(\alpha)$ is a formula as well.*
4 *If α and β are formulae, then $X\alpha$ and $\alpha U \beta$ are formulae as well.*

We assume the following standard abbreviations:

disjunction	$\alpha \vee \beta \equiv_{def} \neg(\neg\alpha \wedge \neg\beta)$
implication	$\alpha \rightarrow \beta \equiv_{def} \neg\alpha \vee \beta$
globally α	$\Diamond(\alpha) \equiv_{def} \top U \alpha$
future α	$\Box(\alpha) \equiv_{def} \neg\Diamond(\neg\alpha)$
permission	$P_{a,b}(\alpha) \equiv_{def} \neg O_{a,b}(\neg\alpha)$
prohibition	$F_{a,b}(\alpha) \equiv_{def} \neg P_{a,b}(\alpha)$
obligation	$O_a(\alpha) \equiv_{def} O_{a,a}(\alpha)$

We now illustrate how to use the logic to reason about assertions. We assume the following propositions: create(c) holds when object c is created, destruct(c) holds when object c is destructed, call(c_1,c_2,f) holds when object c_1 calls routine f in object c_2. We assume that if a routine in an object is called, there is an earlier moment in time at which the object is created. However, since our operators only consider the future, this property cannot be formalized. We assume that propositions can deal with integers, a well known issue in specification and verification, see [14] for further details. Finally, we assume that the time steps of the temporal model are calls to routines. The first routine and the invariant in the class account in Figure 1 can now be formalized as:

call(c_1,c_2,deposit(sum:INTEGER)) $\rightarrow O_{c_1,c_2}$(sum $\geq$ 0)
(call(c_1,c_2,deposit(sum:INTEGER)) $\wedge$(sum >= 0) $\wedge$ (balance = b)) $\rightarrow XO_{c_2,c_1}$(balance = b + sum)
create(c) $\rightarrow$ (O_c(balance $\geq$ min_balance) U destruct(c))

These formulas can be read as follows. If there is a call of c_1 to c_2 to deposit a sum, then c_1 is obliged towards c_2 that this sum is not negative. If there is such a call, the sum is not negative and the balance is b, then there is an obligation of c_2 towards c_1 that the new balance is b increased with the deposited sum. Once an object is created and until it is destructed, it is obligatory that the balance is at least the minimal balance.

2.3 Contract violations

Whenever there is a contract, the risk exists that someone will break it. This is where exceptions come in the design by contract theory. Exceptions – contract violations – may arise from several causes. One is an assertion violation, if run-time assertion monitoring is selected. Another is a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow, or an attempt to create a new object when there is not enough memory available. Unless a routine has been specified to handle exceptions, it will **fail** if an exception arises during its execution. This in turn provides one more source of exceptions: a routine that fails triggers an exception in its caller.

A routine may, however, handle an exception through a **rescue** clause. An example using the exception mechanism is the routine *attempt_deposit* that tries to add *sum* to *balance*:

```
attempt_deposit(sum:INTEGER) is
 local
  failures: INTEGER
 require
  sum >= 0;                          -- Ocr(sum >= 0)
 do
  if failures < 50 then
   add(sum);
   successful := True
  else
   successful := False
 rescue
  failures := failures + 1;
  retry
 ensure
  balance = old balance + sum        -- Orc(balance = old balance + sum)
 end
```

The actual addition is performed by an external, low-level routine *add*; once started, however, *add* may abruptly fail, triggering an exception. Routine *attempt_deposit* tries the deposit at most 50 times; before returning to its caller, it sets a boolean attribute *successful* to *True* or *False* depending on the outcome. This example illustrates the simplicity of the mechanism: the **rescue** clause never attempts to achieve the routine's original intent; this is the sole responsibility of the body (the **do** clause). The only role of the **rescue** clause is to clean up the objects involved, and then either to fail or to retry.

The principle is that *a routine must either succeed or fail*: it either fulfills its contract, or not; in the latter case it must notify its caller by triggering an exception. The

optional **rescue** clause attempts to "patch things up" by bringing the current object to a stable state (one satisfying the class invariant). Then it can terminate in either of two ways: either the **rescue** clause may execute a **retry** instruction, which causes the routine to restart its execution from the beginning, attempting again to fulfil its contract, usually through another strategy (this assumes that the instructions of the **rescue** clause, before the retry, have attempted to correct the cause of the exception), either the **rescue** clause does not end with a **retry** and the routine fails; it returns to its caller, immediately triggering an exception (the caller's **rescue** clause will be executed according to the same rules).

In our design language, the exception can be formalized as a violation, and the exception handler gives rise to a so-called contrary-to-duty obligation, a kind of obligation comes in force only in sub-ideal situations. The formalization of contrary-to-duty obligations has been the subject of many debates in deontic logic due to its role in many of the notorious deontic paradoxes such as the Chisholm and Forrester paradox; we do not go into the details here.

For example, there is a violation if the postcondition does not hold, i.e., we do not have balance = old balance + sum. In case of violation, a retry means that the obligation persists until the next time moment. We extend the language with the proposition *retry*. Now, the fact that a retry implies that the postcondition holds again for the next moment can be characterized as follows: $O_{c_1,c_2}(\phi) \wedge \neg\phi \wedge \mathit{retry} \rightarrow XO_{c_1,c_2}(\phi)$. This formula can be read as follows. If c_1 is obliged towards c_2 that ϕ, ϕ is not the case and retry is true, then in the next state there is again such an obligation for c_1 towards c_2.

3 Contracts for agents

In this section we adapt the design by contract theory to deal with the autonomy of agents, and we extend the deontic design language with preferences.

3.1 Preferences over outcomes

In this paper we are in particular interested in contracts with agent routines [17]. We assume as usual that the distinction between agents and components or objects is that agents are autonomous. In this paper we interpret this autonomy in the sense that agent routines can select among various outputs satisfying the caller's condition. We illustrate our notion of autonomy by adapting the class `Account`, which is often used to illustrate design by contract and other object-oriented techniques, such that a call to a routine may result in several outcomes. An account now consists of a set of bank notes, and when depositing we have to specify not only the amount but also how the amount is distributed over the notes. Moreover, when withdrawing money, the routine can choose how to return it. For example, when returning euro 100 the routine can either return one euro 100 note, two euro 50 notes, five euro 20 notes, etc.

Considering now such an autonomous routine, both routine and caller have preferences over outcomes. The routine specifies which outcomes it tries to achieve, and the caller has preferences over outcomes too, and uses them to evaluate whether the routine has satisfactorily fulfilled the contract. In some cases the preferences of both caller and

routine coincide. For example, concerning the level of precision, both caller and routine may prefer more precise outcomes over less precise ones. However, this is not always the case. For example, a routine may prefer fast global results over slow detailed results.

In the running example, it may seem unnatural to define preferences over outcomes – it is therefore also not a good example to illustrate the use of autonomy for agents. However, many examples discussed in the agent literature can naturally be described in this way. That is, the autonomy of agents can often be described by their ability to decide for themselves which answer to return from a set of alternatives. For example, an agent component for hotel search in a web-based booking application has to choose among a huge set of answers. This agent component can be specified by a contract as defined in section 2. The preconditions may be the location of the hotel, the arrival and departure dates of the customer. An informal postcondition for the hotel search component can be "the component will produce a set of hotels satisfying the precondition". However, among this set of hotels, the caller of the routine may choose only the cheapest hotels. Or the agent component may prefer not to have all the hotels satisfying the preconditions, but to obtain the result in less than one second to economize resources. When these criteria are taken into account in the agent component's preconditions, then the component would not longer be autonomous. However, this is clearly not how it works in practice. The reason that such web services are autonomous is that the number of possible answers is very large, and it is changing all the time. Obliging the caller to foresee all possible answers is unrealistic.

We do not want to claim that all kinds of autonomy – or all kinds of agents - can be modelled using preferences over outcomes. For example, another kind of autonomy is the ability of agents to violate norms. It is not clear how to specify this kind of autonomy using preferences over outcomes. However, this kind of norm autonomy can already be specified in the deontic design language introduced in the previous section, because agents can violate the obligations.

3.2 Quality of outcomes

In the design by contract theory, such preferences have not been incorporated yet. The reason is that this theory has been developed for passive objects and components. However, such preferences have been studied in cases where the routines are more autonomous, such as service level architectures, agent theory and artificial intelligence. We therefore propose to extend the contracts between caller and routine such that the contract specifies the preferences of the routine as well as the preferences of the caller.

In our deontic design language, we have to combine the deontic notion of obligation with conditional preferences studied in practical reasoning and decision theory. We use a preference order on the possible answers given by the component. For instance, consider the `withdraw` routine of the `Account` class. Suppose that the routine can return euro 100 notes, euro 50 notes and euro 20 notes. The routine may prefer to deliver as many euro 100 notes as possible, thereafter as many euro 20 notes as possible and finally as many euro 50 notes as possible. Using 20, 50 and 100 as propositional variables with the obvious meaning, the preference order over outcomes for the routine will be (these outcomes are mutually exclusive): $100 \wedge \neg 50 \wedge \neg 20 <_r \neg 100 \wedge \neg 50 \wedge 20 <_r \neg 100 \wedge 50 \wedge \neg 20$.

Those preferences are given *ceteris paribus* [18], i.e., the routine prefers delivering as many euro 100 notes as possible to delivering as many euro 50 notes as possible all else being equal. Notice that the previous preference order over outcomes can be *conditional*. The conditions are some properties of the input of the component, as properties of the outcomes are used in the preference order. For instance, the routine may use this order only if the sum to be withdrawn is more than 200 euros. In the contrary case, the routine may prefer to deliver as many euro 50 notes as possible. Like in the CP-net formalism [19], the preference specification of the `withdraw` routine can now be represented by *a conditional preference table*:

200+	$100\wedge\neg 50\wedge\neg 20 <_r \neg 100\wedge\neg 50\wedge 20 <_r \neg 100\wedge 50\wedge\neg 20$
$\neg$ 200+	$\neg 100\wedge 50\wedge\neg\ 20 <_r \neg 100\wedge\neg 50\wedge 20 <_r 100\wedge\neg 50\wedge\neg 20$

The caller of the routine may also specify some preference order over the outcomes. For instance, a user of the `withdraw` routine may specify that he/she prefers to have as many euro 20 notes as possible, then to have as many euro 50 notes as possible and finally to have as many euro 100 notes as possible: $\neg 100\wedge\neg 50\wedge\ 20 <_c \neg 100\wedge 50\wedge\neg 20 <_c 100\wedge\neg 50\wedge\neg 20$.

In a preference specification, the caller of the routine may use an "aspiration level" to specify under which level the answer of the component is no more acceptable. For instance, let us resume the previous preference specification for the caller of `withdraw`. The caller may want to precise that in this specification, he/she will consider that the quality is not satisfactory if the `withdraw` routine delivers as many euro 100 notes as possible. This specification does not interfere with the primary preference specification and the caller may be able to change the acceptability level. For instance, he/she may now want to consider only $\neg 100\wedge\neg 50\wedge\ 20$ as a satisfactory quality. We will use a marker $\ll_c$ in the caller preference specification to indicate where the least acceptable outcome is for the caller. This can be viewed as a *quality* specification for the caller of the routine. As previously, we can use conditional preference tables to represent the caller preferences. A complete preference specification of routine `withdraw` is:

200+	$100\wedge\neg 50\wedge\neg 20 <_r \neg 100\wedge\neg 50\wedge 20 <_r \neg 100\wedge 50\wedge\neg 20$
$\neg$ 200+	$\neg 100\wedge 50\wedge\neg\ 20 <_r \neg 100\wedge\neg 50\wedge 20 <_r 100\wedge\neg 50\wedge\neg 20$

$\top$	$\neg 100\wedge\neg 50\wedge\ 20 <_c \neg 100\wedge 50\wedge\neg 20 \ll_c 100\wedge\neg 50\wedge\neg 20$

3.3 Deontic design language

We now extend the syntax of O_{LTL} to OP_{LTL} which takes into account the preference specification. The crucial question here is how time and preferences interact. Can we reason about the change of preferences in time (external dynamics), or can we reason about preferences among propositions at distinct moments in time (internal dynamics)? It is tempting to define temporal preference logics along these lines, but they seem to be too complex to be used in practice. We therefore encode in our logic preferences separately from the temporal reasoning over obligations. The preferences specify desired behavior, but the preferences themselves cannot change. This may seem very limited at first sight, though it should be observed that it is in line with standard models in decision theory, where typically a utility function is assumed to be fixed over time.

The preference relations $<_{a,b}$ are indexed by two objects. The first one represents the object asking for a specification and the second one represents the object on which the preference specification is made. For instance $<_{withdraw,withdraw}$ represents a preference specification on the `withdraw` routine emitted by the routine itself. $<_{c,withdraw}$ represents a preference specification on the `withdraw` routine emitted by another agent or routine C.

Definition 2 (Syntax OP_{LTL}). *Given a finite set A of objects (or components, or agents) and a countable set P of primitive proposition names. The admissible formulae of OP_{LTL} are recursively defined by:*

1 *If Φ is a formula of O_{LTL}, then Φ is a formula of OP_{LTL}.*
2 *If $\alpha, \beta_1, \ldots \beta_m$ are propositional formulae and $a, b \in A$, then the following formula is a formula of OP_{LTL}.*
$\alpha : \beta_1 <_{a,b} \ldots <_{a,b} \beta_i \ll_{a,b} \beta_j <_{a,b} \ldots <_{a,b} \beta_m$

The semantics of the O_{LTL}part of $OP_{LTL}$is straightforward, see for instance [15]. The preference specification semantics is given by CP-net semantics, see [19]. Notice that, as shown in [19], we can use indifference between outcomes in the preference specification without losing interesting properties of CP-nets.

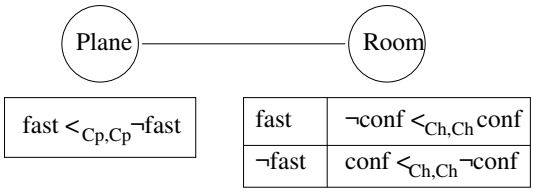
3.4 Contract violations

Now, as a routine contract can provide a level of acceptability in the preference specification expressed by the caller of a routine, we have to define what happens if this acceptability level is not verified by the routine's outcome. For instance, if the `withdraw` routine specified in the previous section delivers as many euro 100 notes as possible, the user specification is violated. We must integrate the acceptability notion into the contract we defined previously. Let us consider a routine r, its preconditions $O_{cr}(\phi)$ and a preference specification $<_{c,r}$ and its associated quality level represented by $\beta_1 <_{c,r} \ldots \beta_j \ll_{c,r} \ldots <_{c,r} \beta_m$. There are two possibilities:

– either the violation of the acceptability level is unacceptable for the caller and in this case we can express it as a postcondition for the component. This will be called as a **strong** acceptability level. We can integrate the acceptability level in the contract by specifying $\phi \rightarrow XO_{rc}(\beta_1 \vee \ldots \vee \beta_j)$.
– either the violation of the acceptability level is acceptable for the caller. For instance, a caller may consider that what is important for him/her is that the component produces an outcome verifying the postcondition. The quality specification he/she produces is a bonus for his/her use of the application. In this case, we cannot express the acceptability level as a postcondition, because the violation of the postcondition will induce strong consequences on the component. We denote such acceptability level specification as **weak** acceptability specification. It can be integrated in the contract by the following formula: $\phi \rightarrow X(\neg(\beta_1 \vee \ldots \vee \beta_j) \rightarrow \textit{unsatisfied}(c))$. The meaning of *unsatisfied* is the following: if the quality is not enough for the caller, then he/she has a right which he/she can execute or leave.

In the case of a strong acceptability specification, there are still good reasons to differentiate the satisfactory quality and "classical" postconditions. The acceptability specification can evolve: the user can change his/her mind, there is not only one user, ..., so the acceptability specification is not a real postcondition which will be verified by all "executions" of the component.

In a real-world application, several components are combined in order to build the whole application. Those components will have contracts as preference specification. We can use the CP-net formalism to represent the information flow among components and to represent the quality specification of a component as preference relations conditioned by the outcome of the previous component. For instance, consider a web-based booking system. This system is composed of two components: a component C_p which searches plane tickets and a component C_h which searches hotel rooms. A user of the system can specify that he/she wants first to find a plane ticket before finding an hotel. For instance, the first component may prefer fast travels. C_h may then prefer cheap hotels if the outcomes of C_p are fast travels (because they are more expensive), and comfortable hotels if the travel is not fast (because the traveller may want to have rest). A CP-net graph formalising this specification is:



Using the CP-net machinery, we can deduce that the preference specification for the global component is: fast∧¬conf $<_{Cp+Ch,Cp+Ch}$ fast∧conf $<_{Cp+Ch,Cp+Ch}$ ¬fast ∧ conf $<_{Cp+Ch,Cp+Ch}$ ¬fast∧¬conf. An important subject for further research is how to formally derive global acceptability level from each component's acceptability level, or the implication of using a cyclic graph representing the components "communications". Some references about cyclic CP-nets are given in [19].

4 The normative stance

In this section we compare the normative stance, a phrase due to Jan Broersen [20] and implicit in design by contract, with the intentional or BDI stance popular in agent oriented software engineering. The following table summarizes the comparison between the intentional stance and the normative stance:

Stance	intentional stance	normative stance
Concepts	BDI	OP, rights, responsibility
from	folk psychology	ethics, law, sociology
Computer	human = angry, selfish, ...	God, master/slave, servant
Class of systems	decision making	decision making
Realization	specification and verification	components
Implementation	programming	objects, operation
specification	BDI_{CTL}	temporal deontic logic

First, the intentional stance is rooted in the philosophical work of Dennett, whereas such grounding does not seem to exist for the normative stance (though there are candidates, such as [21]). The concepts from the intentional stance come from folk psychology. The normative stance borrows concepts from ethics, law or sociology. Other examples of this normative stance we mentioned in the introduction are the IMPACT system [7] and the BOID architecture [8].

Second, the success of the intentional stance is that people like to talk about their computer as a human which has beliefs and desires, which may be selfish, or which can become angry. The implicit assumption of design by contract is that designers find it useful to understand software construction in terms of contracts, or, more generally, in terms of obligations. The success of design by contract may be explained by the fact that "social contract" is well established in social sciences [22]. We may call this the normative stance towards computer systems. The success is due to the fact that humans either consider the computer as their master, which has to be obeyed, or as their slave, which has to obey orders.

Third, the intentional stance has been advocated for agent systems, which are for example autonomous and proactive. It has been used as a high level specification language, as well as low level programming language. We believe the normative stance can be used in a wider setting. In the examples we used it also for low level objects. However, it is particularly useful if we use a higher abstraction level in terms of components or agents.

5 Concluding remarks

In this paper we study how extensions of deontic logic can support design. We propose a deontic design language, that is a kind of specification language whose primary operator is an "obligation" operator (see Section 4). First, we ask which kind of deontic logic can be used as a design language to support design by contract We show how directed modal operators are capable of formalizing contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by assertions. These formalisms have been developed and studied in electronic commerce and artificial intelligence and law. Moreover, we show how temporal operators can be used to formalize dynamic behavior such as contract violations.

Second, we introduce preferences over outcomes of a routine. This is a necessary extensions of the design by contract approach when the components is autonomous in the sense that it can return several outputs, such as autonomous agents or autonomous services. We illustrate how the preferences can be used to specify the desired quality of a contract. We show how the preferences can be specified with ceteris paribus or CP nets. In further research we study qualities of service level contracts that refer to multiple routine calls, such as average response times.

Third, we ask what kind of properties should be formalized by such a design logic. This is summarized in the following table:

social contract	assertions	directed obligations
violation	exception	violations
repair	exception handling	contrary-to-duty reasoning
contract form	interface	?
testing and debugging	?	?

In this paper, we do not consider contract forms and contracts for testing and debugging. The contract form of a class, also called its "*short form*", serves as its interface documentation. It is obtained from the full text by removing all non-exported features and all implementation information such as **do** clauses of routines, but keeping interface information and in particular assertions. The use of these elements in our deontic design language, for example to *combine* assertions, is subject of further research.

Fourth, we ask how this approach based on deontic logic compares to the BDI approach, dominant in agent based software engineering. Whereas the BDI approach is based on an attribution of mental attitudes to computer systems, design by contract is based on an attribution of deontic attitudes to systems. We suggest that the normative stance has a wider scope of applicability than the intentional stance, though this has to be verified in practice. In further research we study the relation with commitments in Shoham's Agent Oriented Programming (AOP) [17], and with rely/guarantee reasoning [23].

The formalism developed here may seem too "formal" to be used in real applications. It would be interesting to develop practical tools taking our approach into account, in order to offer a support for deontic software engineering. We may for instance extend CP-nets tools.

Another topic for further research is the introduction of other elements of contracts in our formalism. Contracts typically consist not only of regulative norms (obligations), but also of constitutive norms (counts-as conditionals) [24]. How to introduce them in design by contract, and in particular in our deontic design language $\mathrm{OP}_{\mathrm{LTL}}$?

References

1. Meyer, B.: Design by contract. In Mandrioli, D., Meyer, B., eds.: Advances in Object-Oriented Software Engineering. Prentice-Hall, New York, London (1991) 1–50
2. Meyer, B.: Applying design by contract. IEEE COMPUTER **25(10)** (1992) 40–51
3. Meyer, B.: Systematic concurrent object-oriented programming. Communication of the ACM **36(9)** (1993) 56–80
4. Meyer, J., Wieringa, R.: Deontic Logic in Computer Science: Normative System Specification. John Wiley and Sons (1993)
5. von Wright, G.: Deontic logic. Mind **60** (1951) 1–15
6. Wieringa, R., Meyer, J.: Applications of deontic logic in computer science: A concise overview. In: Deontic Logic in Computer Science. John Wiley & Sons, Chichester, England (1993) 17–40
7. Eiter, T., Subrahmanian, V., Pick, G.: Heterogeneous active agents, I: Semantics. Artificial Intelligence **108** (1999) 179–255
8. Broersen, J., Dastani, M., Hulstijn, J., van der Torre, L.: Goal generation in the BOID architecture. Cognitive Science Quarterly **2(3-4)** (2002) 428–447

9. Meyer, B.: Invitation to Eiffel. Technical Report TR-EI-67/IV, Interactive Software Engineering (1987)
10. Dignum, F.: Autonomous agents with norms. Artificial Intelligence and Law **7(1)** (1999) 69–79
11. Krogh, C., Herrestad, H.: Hohfeld in cyberspace and other applications of normative reasoning in agent technology. Artificial Intelligence and Law **7(1)** (1999) 81–96
12. Singh, M.P.: An ontology for commitments in multiagent systems: toward a unification of normative concepts. Artificial Intelligence and Law **7** (1999) 97–113
13. Tan, Y., Thoen, W.: Modeling directed obligations and permissions in trade contracts. In: Proceedings of the Thirty-First Annual Hawaian International Conference on System Sciences. (1998)
14. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, Heidelberg, Germany (1992)
15. Broersen, J., Dastani, M., van der Torre, L.: BDIO_{CTL}: Properties of obligation in agent specification languages. In: Proceedings of IJCAI'03. (2003) 1389–1390
16. Jamroga, W., van der Hoek, W., Wooldridge, M.: On obligations and abilities. In: Deontic logic in computer science. Volume 3065 of LNAI. (2004) 165–181
17. Shoham, Y.: Agent-oriented programming. Artificial Intelligence **60** (1993) 51–92
18. Boutilier, C., Brafman, R., Hoos, H., Poole, D.: Reasoning with conditional ceteris paribus preference statement. In Laskey, K., Prade, H., eds.: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann (1999) 71–80
19. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H., Poole, D.: CP-nets: a tool for representing and reasoning with conditional *ceteris paribus* preference statements. Journal of Artificial Intelligence Research (JAIR) **21** (2005) 135–191
20. Broersen, J.: Modal Action Logics for Reasoning about Reactive Systems. PhD thesis, Vrije Universiteit Amsterdam (2003)
21. Brandom, R.: Making it explicit. Harvard University Press, Cambridge, MA (1994)
22. Rousseau, J.: The social contract. (1762) http://www.constitution.org/jjr/socon.htm.
23. Stark, E.W.: A proof technique for rely/guarantee properties. In: Foundations of Software Technology and Theoretical Computer Science. Volume 206 of Lecture Notes in Computer Science. (1985) 369–391
24. Boella, G., van der Torre, L.: Contracts as legal institutions in organizations of autonomous agents. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'04). (2004) 948–955